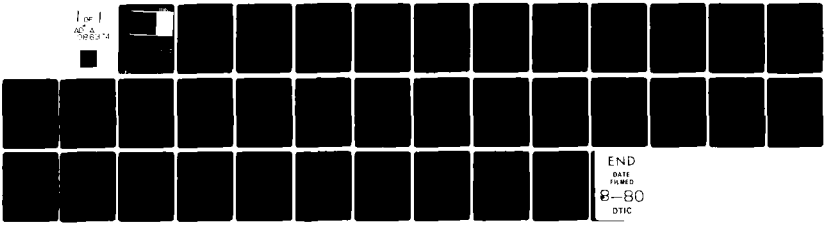
AD-A086 374 WISCONSIN UNIV-MADISON MATHEMATICS RESEARCH CENTER F/G 9/2
DESIGN CONSIDERATIONS FOR DATA-FLOW DATABASE MACHINES.(U)
MAR 80 H BORAL, D J DEWITT DAAG29-79-C-0165
UNCLASSIFIED MRC-TSR-2058 NL



END
DATE FILMED
8-80
DTIC

MRC Technical Summary Report # 2058

DESIGN CONSIDERATIONS FOR DATA-FLOW DATABASE MACHINES

Haran Boral and David J. DeWitt

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

March 1980

Received February 22, 1980

Approved for public release
Distribution unlimited

Sponsored by

U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

National Science Foundation
Washington, D. C. 20550

UNIVERSITY OF WISCONSIN- MADISON
MATHEMATICS RESEARCH CENTER

DESIGN CONSIDERATIONS FOR DATA-FLOW DATABASE MACHINES.

Haran Boral [redacted] David J. DeWitt

Technical Summary Report # 2058
March 1980

ABSTRACT

This paper presents a discussion of the application of data-flow machine concepts to the design and implementation of database machines which execute relational algebra queries. We analyze the performance of multiprocessor nested-loops and sort-merge join algorithms and show that the nested-loops algorithm is generally superior. Three levels of operand granularity for data-flow database machines are introduced and compared using the nested-loops join algorithm. We demonstrate, that relation-level granularity is too coarse and that tuple-level granularity is too fine. The third level of granularity, a page of a relation, is shown to be the best choice from both hardware and software viewpoints. Finally a preliminary design for a data-flow database machine which utilizes page-level granularity and supports distributed control of instruction execution is presented.

AMS(MOS) Subject Classification: 68A50

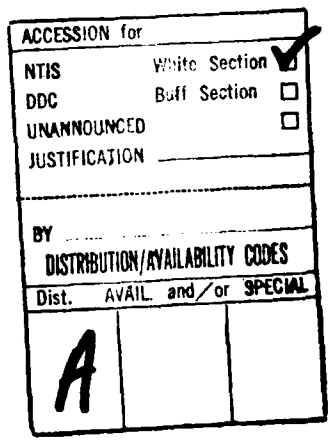
Key Words: Database management systems, relational databases, database machines, associative memories, parallel processing, query execution and optimization, dataflow machines.

Work Unit No. 3 - Numerical Analysis and Computer Science

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024 and No. DAAG29-79-C-0165 and the National Science Foundation under grant MCS78-01721.

SIGNIFICANCE AND EXPLANATION

A preliminary architecture of a database machine is presented. This architecture uses the presence of data as the criterion for instruction initiation and control rather than the instruction's position in the program. Such a machine is known as a data-flow machine. The use of data-flow techniques in database machines has been shown to be promising elsewhere. However, to this date no architecture using these techniques has been designed. The architecture presented is only a preliminary one and will most likely undergo a number of future changes.



The responsibility for the wording and views expressed in this descriptive summary lies with MRC, and not with the authors of this report.

DESIGN CONSIDERATION FOR DATA-FLOW DATABASE MACHINES

Haran Boral and David J. DeWitt

1. Introduction

During the past several years we have been investigating the design and implementation of multiprocessor database machines for the execution of relational algebra queries. In [1,2] the architecture of DIRECT, a MIMD database machine, is described. The problem of relation fragmentation and its impact on query execution time is discussed in [3]. In [4], four processor assignment strategies for MIMD database machines are described and evaluated. One of the primary results presented in [4] is that the application of data-flow machine techniques to the processing of relational algebra queries significantly enhances system performance. The architecture of DIRECT [1,2] is that of a data-flow machine where all the control functions are centralized. In this paper we intend to present an approach to constructing a data-flow database machine which supports distributed control.

In Section 2.0, we introduce the basic concepts of query processing using multiple processors and data-flow machines. We analyze the performance of multiprocessor nested-loops and sort-merge join algorithms and show that the nested-loops algorithm is generally superior. In Section 3.0, three levels of operand granularity for data-flow database machines are introduced and compared. We demonstrate, that relation-level granularity is too coarse and that tuple-level granularity is too fine. The third level of granularity, a page of a relation, is shown to be the best choice from both hardware and software viewpoints. Section 4.0 contains a preliminary design for a data-flow database machine which supports page-level granularity. Our conclusions

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024 and No. DAAG29-79-C-0165 and the National Science Foundation under Grant MCS78-01721.

and areas of future research are presented in Section 5.0.

2. Background

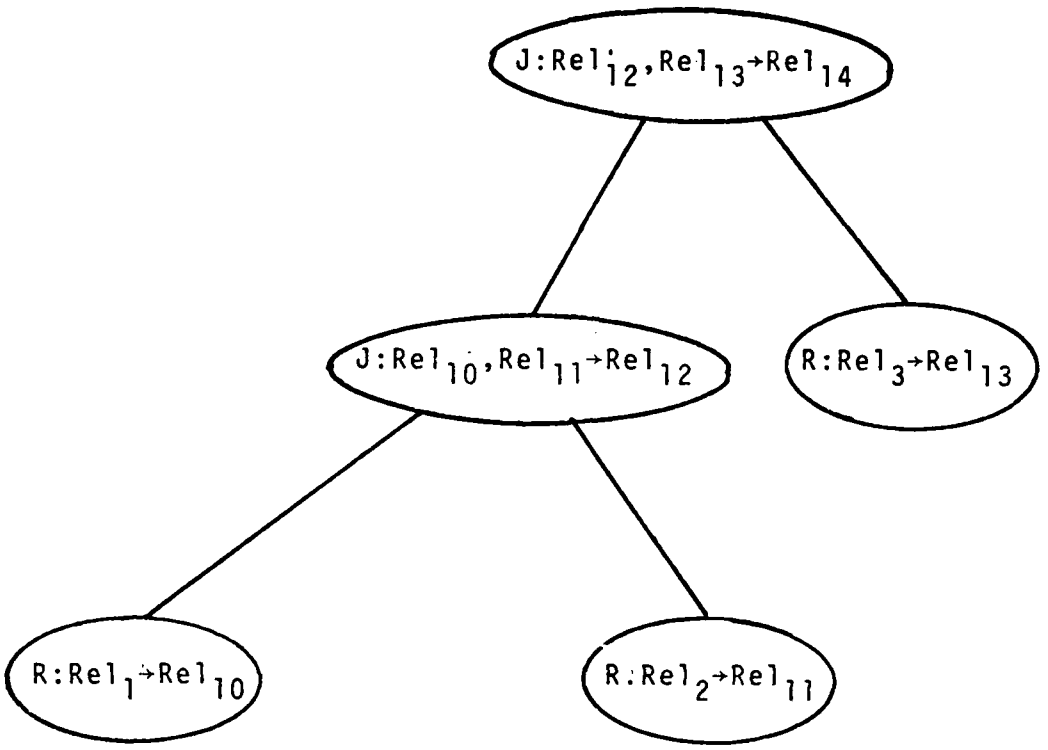
2.1. Relational Query Processing

Each relational algebra query is generally comprised of one or more relational algebra operations (instructions) and is organized in the form of a tree. Each node, represents an operation to be performed on a number of relations. Some examples are restrict, join, append, and delete. Nodes higher up in the tree operate on relations produced by nodes below them. Figure 2.1 contains an example of a typical relational algebra query in the form of a query tree.

2.1.1. Parallel Join Algorithms

In a relational database system one of the most time consuming operations that must be performed is the join operator. In [5], several alternative join algorithms for uniprocessor systems are presented and analyzed. The results show that in the absence of indices (as in DIRECT) a sort-merge algorithm performs best. However, for multiprocessor systems we feel that the performance of the nested-loops algorithm is superior to that of the sort-merge algorithm. To support this claim we present some intuitive arguments followed by a short, informal analysis of both algorithms.

The multiprocessor sort-merge algorithm employs a parallel sort of both relations on the joining attribute. This is followed by a uniprocessor merge on the joining attribute to perform



R:Restrict
J:Join

Figure 2.1

A Sample Query Tree

the join. Our parallel sort uses a binary tree arrangement of processors as in [6].

The multiprocessor nested-loops algorithm works by joining each unit of one (the outer) relation with all of the units in the other (the inner) relation. If a unit corresponds to a page, the outer relation is n pages long, and there are n processors available, then each processor can join one page of the outer relation with the entire inner relation. A unit can also be a tuple.

We assume that each page of both relations is sorted and that a merge algorithm is used by the multiprocessor sort-merge algorithm to sort two pages and by the multiprocessor nested-loops algorithm to join two pages. Therefore, the time required for a processor to process two input pages is the same for both algorithms and can be disregarded in the following comparison of the two algorithms. Finally, for the sorting analysis we have made a number of simplifying assumptions. The most significant is that after each stage all the processors flush their buffers. Although optimizations will improve the total execution time the improvement will not be significant. Disregarding optimizations makes the analysis easier.

Intuitively the nested-loops algorithm should outperform the sort-merge since the amount of parallelism that can be attained is high (limited only by the number of pages in the outer relation) and can be maintained throughout the duration of the execution. When sorting in parallel one may be able to start with a large number of processors but after each stage the number of

processors decreases whereas the amount of data examined by each processor increases until in the final stage one processor must examine the relation in its entirety.

A second consideration that is not immediately apparent is that of cache memory usage. In order for the parallel sort to execute optimally the complete relation must be kept in the cache for the duration of the sort. In the case of a nested-loops join, only a portion of the inner relation must be kept in the cache for the duration of the operation (since once a page of the inner relation has been seen by all the processors that page frame can be used for another page of that relation).

The following informal analysis verifies our intuitive arguments about the relative performance of the two algorithms. We assume that the two relations to be joined contain n and m pages and that $n > m$. We also assume that there are p processors, $1 \leq p \leq n$, available to perform the join and each processor has a 3 page internal buffer: 2 for input and 1 for output. To simplify the analysis we have also assumed that p,m,n are all powers of 2.

For the multiprocessor nested-loops algorithm, the execution time of the join, $t_{nested\text{-}loops}$, is equal to $n/p*(1+m)$. Each of the p processors will each join n/p pages of the outer relation with all m pages of the inner relation. Thus each of the processors will read 1 page of the outer relation followed by m pages of the inner relation. This will occur n/p times.

For the multiprocessor sort-merge algorithm, the execution time of the join, $t_{sort\text{-}merge}$, is equal to $t_{sort}(outer) + t_{sort}(inner) + t_{merge}$, where $t_{merge} = n + m$. The sort time for

each relation, t_{sort} equals $t_{suboptimal} + t_{optimal}$, where

(*) $t_{suboptimal} = 2n/p*[\log_2(n/2p)+1]$

(**) $t_{optimal} = 4n-2n/p$.

The derivation of (*) follows. At each suboptimal level the p processors do the same amount of work: 2n/p I/O operations (n/p reads and n/p writes). At level i, i>0, each processor sees 2 "runs" whose length is 2^{i-1} pages and produces one run whose length is 2^i pages. The first optimal level is that level whose input runs are each of length n/2p (thus each of the p processors inputs exactly n/p pages). This is level number $\log_2(n/2p)+1$. Since we start counting levels from 0 there are $\log_2(n/2p)+1$ suboptimal levels.

To derive (**) we note that in each level we do twice as much work as in the preceding level. We start with n/p reads and n/p writes by each processor. There are $\log_2(2p)$ optimal levels. Thus $t_{optimal}$ is the sum of the $\log_2(2p)$ terms 2n/p, 4n/p,...,2n. This sum simplifies to 4n-2n/p.

Our two final formulas are then:

$t_{nested\text{-}loops} = n/p*(1+m)$, and

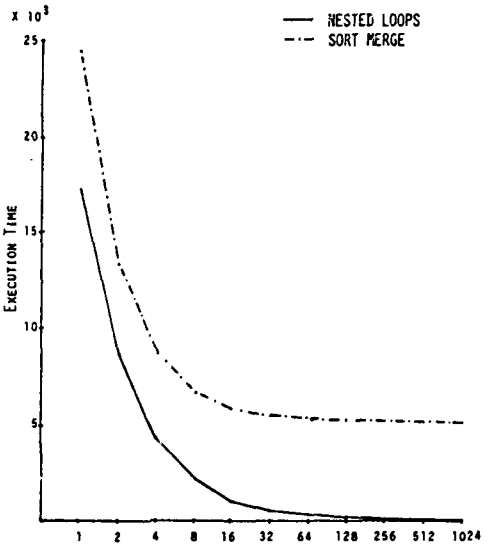
$$\begin{aligned} t_{sort\text{-}merge} &= 2n/p*[\log_2(n/2p)+1]+4n-2n/p \\ &\quad + 2m/p*[\log_2(m/2p)+1]+4m-2m/p \\ &\quad + n+m \\ &= 2n/p*\log_2(n/2p)+2m/p*\log_2(m/2p)+5n+5m \end{aligned}$$

It is clear that as p approaches n, the nested-loops algorithm outperforms the sort-merge. For very small values of p (relative to n) the opposite is true. This is as expected since the uniprocessor sort-merge algorithm has nlogn complexity while

the nest-loops algorithm has n^2 complexity. Figures 2.2-2.4 present the results of the behavior of the two algorithms for three different joins. We see that the nested-loops algorithm generally outperforms the sort-merge algorithm when the number of processors executing the join is a fairly small fraction of the "optimal" number of processors for the nested-loops algorithm (i.e. the number of pages in the larger/outer relation). It should also be noted that at optimal levels the nested-loops algorithm outperforms the sort-merge according to the ratio (1+m)/5*(n+m).

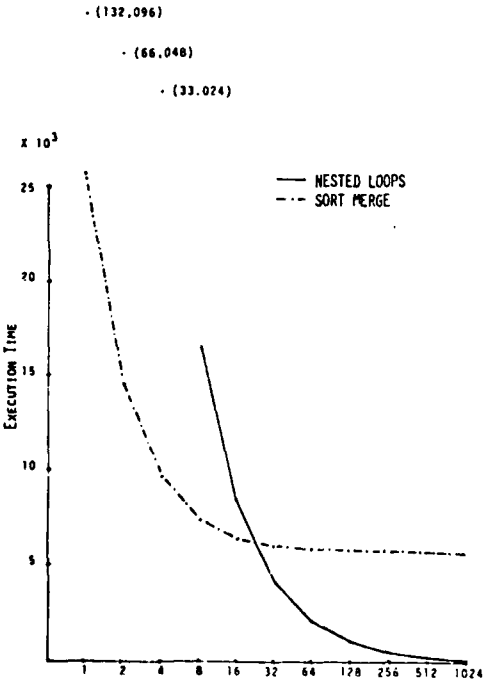
2.1.2. Parallel Update Operations

The following algorithms are employed by DIRECT for the three update operators append, delete, and replace. Deletes are implemented as negated restricts. To perform an append we execute a variation of a merge. We assume that all pages are sorted on either the key or the entire tuple. The tuples to be appended are placed in as few pages as possible in a sorted order. Each processor executing the append is given a page of the original relation and in turn all the new pages. The processor examines both pages and upon finding a duplicate tuple it deletes the tuple from the old page. Finally the new pages are added to the relation page table. Replace is implemented as a modified delete followed by an append. In each case the pages of the result relation remain sorted. Each relation must undergo a periodic reorganization if pages become too sparse.



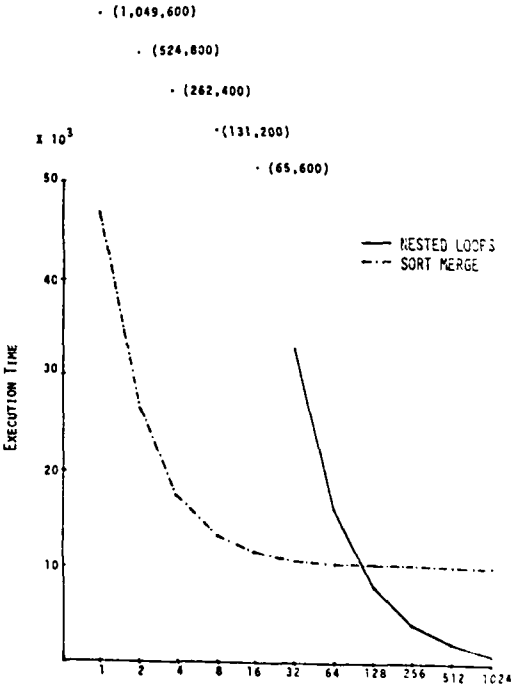
1024 JOIN 16

Figure 2.2



1024 JOIN 128

Figure 2.3



1024 JOIN 1024

Figure 2.4

2.2. Data-flow Machines

A data-flow machine is an architecture devoid of a program counter where instructions are enabled for execution as soon as their operands are present. Such a machine consists of a memory section, a processing section, and an interconnection device between the two sections. A memory cell contains an instruction and room for the operand data. As soon as all the required data is present, the contents of the cell are sent to some processor for execution. This frees the cell for the execution of the next instruction. Output from the processor is sent via the interconnection device to one or more memory cells, possibly enabling one or more instruction(s) in the destination cell(s).

Various architectures for data-flow machines have been proposed [7-11]. These architectures differ from each other in many ways. One difference is the granularity of the operands and the types of operations that the processors execute. For example, Dennis [7] talks about assigning such instructions as add and multiply to the processors whereas Arvind [8] and Rumbaugh [10] assign entire procedures to processors.

For data-flow database machines there are also several alternative variable granularities for enabling relational algebra operators in the query tree. That is, the basic variable used for scheduling decisions can be a whole relation, a fragment of a relation, or a single tuple. In Section 3.0, we will describe and then contrast each of the these granularities.

In order to illustrate our ideas we chose to use the MIT machine [7] as a model since it is easy to understand and

describe. Furthermore we feel that although the model differs significantly from others the basic results remain unchanged. The machine organization of Figure 2.5 depicts the model described in [7]. Although later, more sophisticated, variations have been described in the literature [11] we feel that they do not conceptually differ from the original.

In the machine of Figure 2.5 the interconnection mechanism is divided into two sections. The arbitration network provides a path from every memory cell to every processor. Enabled cells travel through it to processors for execution. Result packets are sent from the processors through the distribution network to the memory.

2.3. Relational Query Processing in Data-Flow Machines

We assume that the instruction in each memory cell corresponds to a node in the query tree and that the data is represented by page tables, pointing to pages either in a mass-storage cache or on mass storage. Thus a relation can also be thought of as a stream [11] of pages. In order to simplify our discussion we assume that at the time that a memory cell fires, the associated data pages are retrieved from a mass-storage cache and placed, together with the control information, on the arbitration network. Similarly, the distribution network places output pages in the mass-storage cache and updates the page tables in the target cells.

The processing of queries in a data-flow fashion is related to the idea of processing relational queries in a pipelined

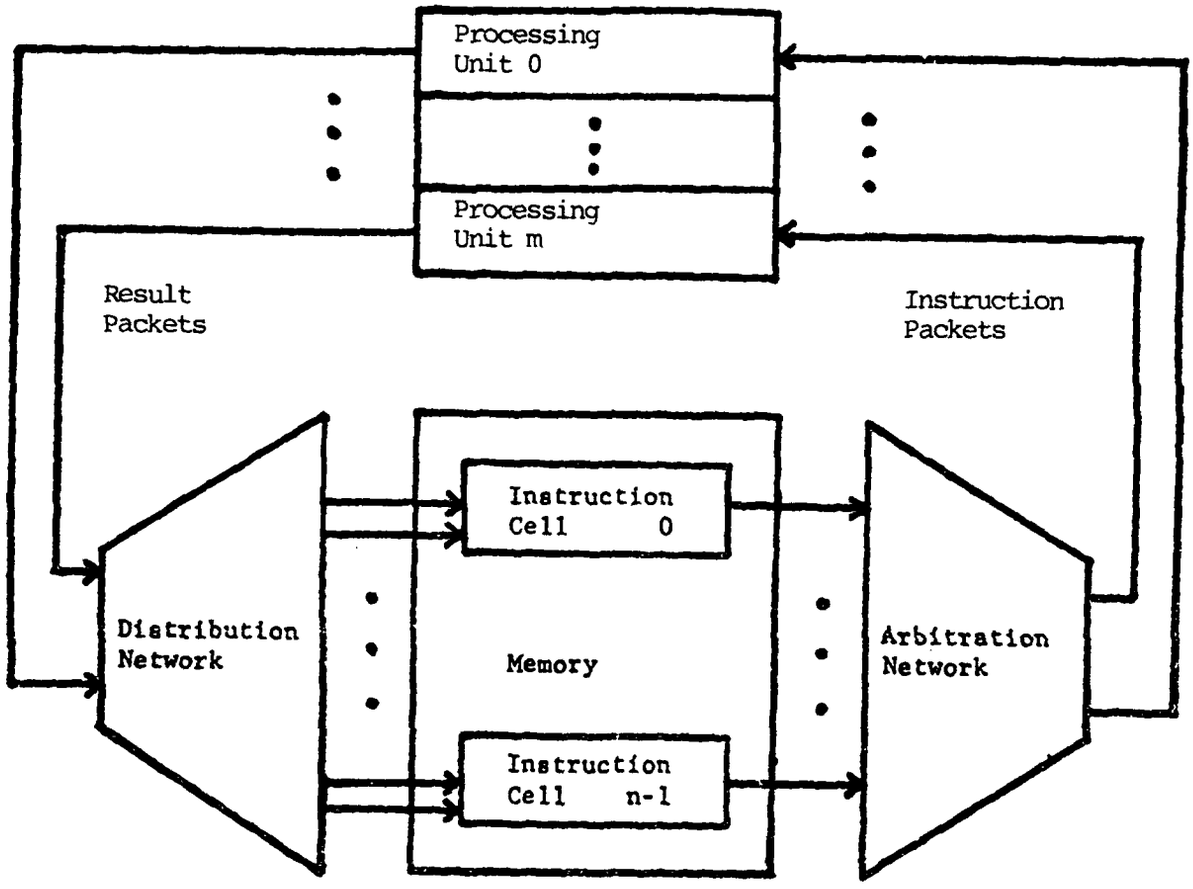


FIGURE 2.5

THE MIT DATA-FLOW MACHINE MODEL

fashion which has been previously suggested by Smith and Chang [12] and Yao [13]. There are, however, several important differences between the two approaches. In the pipelined approach, there will be at most one processor executing each node in the tree and therefore the concurrency obtained will be limited by the number of nodes in the query tree. In the data-flow approach we can have any number of processors executing each node and can dynamically adjust which processors are executing which nodes in the query tree in order to maximize performance. The other major difference is that in the data-flow approach we never need to wait for one node to completely finish before initiating the subsequent operator as has been suggested is necessary for pipelining [13].

3. Three Operand Granularities for Data-flow Query Processing

3.1. Relation-level Granularity

The coarsest possible granularity for enabling instructions is the relation. That is, a node in the query tree is enabled for execution only when its source operands have been completely computed. Clearly, if the query is in a tree format, all leaf nodes are immediately executable. A node higher up in the query tree is enabled whenever all of its descendants have finished executing.

3.2. Page-level Granularity

In this approach a page of a relation is used for scheduling decisions. This means that an operator can be initiated as soon

as at least one page of each participating relation exists. Assigning processors to operate on pages rather than relations makes it possible to cut down on page traffic between the data-flow machine memory and the mass storage device(s) by distributing processors across all nodes of the tree and pipelining pages of intermediate relations between them.

In order to evaluate data-flow query processing which employs relation-level granularity with page-level granularity a detailed simulation of DIRECT was implemented [4]. While this simulation measures the performance of each data-flow strategy on a multiprocessor organization [1,2] which is not a true data-flow machine (i.e. it has centralized control), we feel that similar results would be obtained if the strategies were tested on a machine with more decentralized control organization.

The following assumptions were made:

- 16K byte operands for instruction packets
- LSI-11s as processors (can read a 16K byte page in 33ms)
- The data cache is constructed from Intel 2314 CCD chips
- Two IBM 3330 disk drives for mass storage of relations
- A cross-bar switch with broadcast capabilities is used to connect the processors with the data cache. The cross-bar switch is a feature of DIRECT, not data-flow DIRECT.

In [4] a description of the experiments and results is presented. Figure 3.1 shows the results of the simulation for a representative benchmark containing ten queries (2 queries with 1

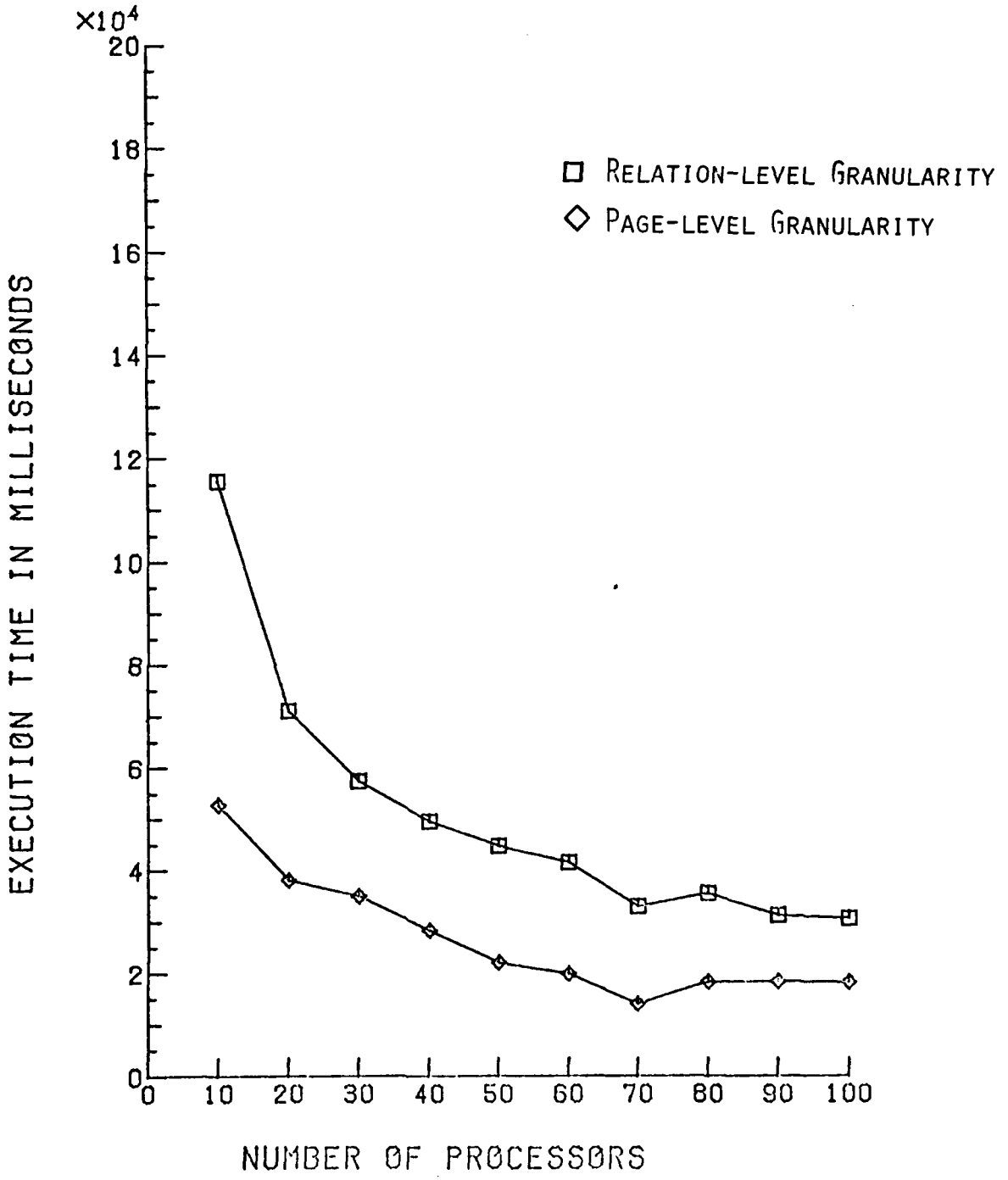


FIGURE 3.1

COMPARISON OF PAGE-LEVEL AND RELATION-LEVEL GRANULARITIES

restrict operator only, 3 queries with 1 join and 2 restricts each, 2 queries with 2 joins and 3 restricts each, 1 query with 3 joins and 4 restricts, 1 query with 4 joins and 4 restricts, and 1 query with 5 joins and 6 restricts), a relational database containing 15 relations with a combined size of 5.5 megabytes, and two CCD cache page frames for each processor. As illustrated by this experiment the page-level granularity outperforms relational-level granularity by a factor of about two to one (The interested reader is referred to [4] for results for different query mixes). These results seem to verify the benefits of pipelining pages of relations up the query tree in order to minimize movement of data between a shared data cache and secondary memory.

3.3. Tuple-level Granularity

In this approach a tuple of a relation is the basic unit which is used for scheduling decisions. This means that an operator can be initiated as soon as at least one tuple of each participating relation exists. As with page-level granularity, this granularity also offers the possibility of pipelining tuples of intermediate relations between nodes in the query tree. However, this granularity places unnecessarily high bandwidth requirements on the arbitration network as will be demonstrated below.

When the nested-loops join algorithm is applied with tuple-level granularity, each tuple of the outer relation will be joined with every tuple of the inner relation. Let the outer

relation be A and the inner relation be B. Assume that the number of tuples in A is n and the number of tuples in B is m. Furthermore, assume that each tuple in A and B is 100 bytes long and that c represents the number of overhead bytes associated with each instruction that passes through the arbitration network. To execute the join, n*m*(200+c) bytes will have to pass from the memory through the arbitration network to the processing section.

Next consider the bandwidth requirements if this same example is executed using page-level granularity. Assume that each page is 1000 bytes long. Therefore, relation A occupies n/10 pages and relation B occupies m/10 pages. Thus n*m*(20 + c/100) bytes must pass through the arbitration network. Even if one ignores the overhead of sending a packet (which is probably the same for both granularities), the bandwidth requirements of the page approach is 1/10 that of the tuple level approach.

While increasing the page size to 10,000 bytes will obviously decrease the arbitration network bandwidth requirements by another order of magnitude, such an increase may have an adverse effect on query execution time because it may reduce the degree of concurrency which is possible. If the number of processors available for query execution is approximately equal to n * m, tuple-level granularity is optimal. We feel that this is unlikely as typically the value of n * m will be in the millions. Therefore for typical queries (unless there are millions of processors), tuple-level granularity places an unnecessary burden on the arbitration network without an apparent increase in

performance. By sending pages of relations to the processors, a similar degree of concurrency can be achieved while minimizing network traffic.

4. A Preliminary Architecture for a Data-flow Database Machine

While the architecture of the MIT machine could be used as the basis of a data-flow database machine, we have identified several properties which for a database machine will unnecessarily limit its functionality and increase its complexity. The MIT machine [7] is designed to permit the simultaneous execution of the instructions from only one program (or one query). This clearly is very restrictive for a multiuser, database management system environment.

Furthermore, we feel that for a database machine the same level of performance can be achieved with an entirely different design for the arbitration and distribution networks. These networks are responsible for instruction initiation and data distribution. The design of the data distribution network is relatively straightforward. Its function is to take a result packet produced by a processor and store it in those instruction cells which are specified in the packet header. The arbitration network, on the other hand, is very complex. It must continuously monitor all instruction cells and provide a mechanism for initiating several enabled instructions simultaneously by routing the contents of each enabled instruction to a free processor for execution. We feel that for data-flow database machines these two networks are too general purpose and consequently excessively

expensive.

In our approach the instruction memory and the arbitration and distribution networks are replaced with a small number of relatively low-performance processors. Each processor will be responsible for controlling the execution of a few (perhaps only one) relational algebra operations. Thus control of the execution of a query is distributed among a set of processors. When an instruction controller (IC) is given a relational algebra operation to control it is also given an initial allocation of processors (called instruction processors - IPs) for executing the instruction. If a typical query contains five operations, then fifty ICs can maintain a multiprogramming level of at least ten in the database machine.

Our approach appears to be viable for two reasons: program size (number of instructions) and execution time of a typical instruction. One frequently mentioned application [7] for data-flow machines is large scientific programs (e.g. weather programs). These programs generally consist of thousands of instructions each of which takes only a few microseconds (or less) to execute. Even if the instruction operates on operands of type vector, multiple processors can be used to work on individual elements and hence instruction execution time will still be in the microsecond range. For these applications a large instruction memory is required to hold the entire program. Since each instruction cell has one input to the arbitration network, the size of the arbitration network is proportional to that of the instruction memory. The arbitration and distribution

networks must also be extremely fast. For example, if 100 one microsecond (execution time of a typical instruction) processing elements are to be kept busy, the arbitration network must be capable of routing 10^8 packets/second.

Relational algebra queries, on the other hand, are composed of relatively few instructions (typically 1-10 operations) each of which takes a relatively long time to execute (in the millisecond to second range). Also packets originating from one IC are sent to a fixed subset of instruction processors, as, for example, are the inner relation pages in the join. This permits us to replace the instruction memory and the two networks with a set of processors without any loss of performance or functionality.

4.1. Hardware Organization and General Operation

In this section we present one possible design for a data-flow database machine. Our purpose in studying this architecture is to enable us to learn more about problems associated with data-flow database machines. This ring-based organization is of course limited by the communication medium bandwidth. However, it will later be shown that bandwidth requirements placed on the ring for a fairly large configuration are not unreasonable. The organization contains six major components:

1) The master controller (MC).
2) A set of instruction controllers (IC).
3) A communications ring (inner ring) which connects the master controller with the instruction controllers.

4) A mass storage system with a multiport disk cache.

5) A set of instruction processors (IP).

6) A communications ring (outer ring) which connects the instruction processors with the instruction controllers.

The MC serves a number of functions. The first is to handle communications with the host processor. When a user's query (in the form of a query tree) is received by the MC it is placed in a queue of queries awaiting execution. When system resources (ICs and IPs) become available, the MC removes the next query from the queue, checks it for concurrency conflicts with other executing queries, and then distributes a subset of the instructions from the query to a set of instruction controllers. The other functions of the MC are to control utilization of the disk cache among the ICs and to control IP allocation.

Each IC is responsible for controlling one or more instructions. Controlling an instruction involves first acquiring a set of IPs from the MC and then distributing instruction packets (see Section 4.2) to the allocated IPs. Thus the ICs compete with each other for the processors in the IP pool. The MC is responsible for arbitration of the requests in a manner which maximizes system performance by insuring that processors are distributed across all nodes in a query tree.

Each IC has a local memory for pages of source relations which will be used as operands in the instruction packets it distributes to the IPs. When the local memory of an IC fills, the IC will write the least desirable pages to the multiport disk cache. One possible approach for controlling usage of the disk

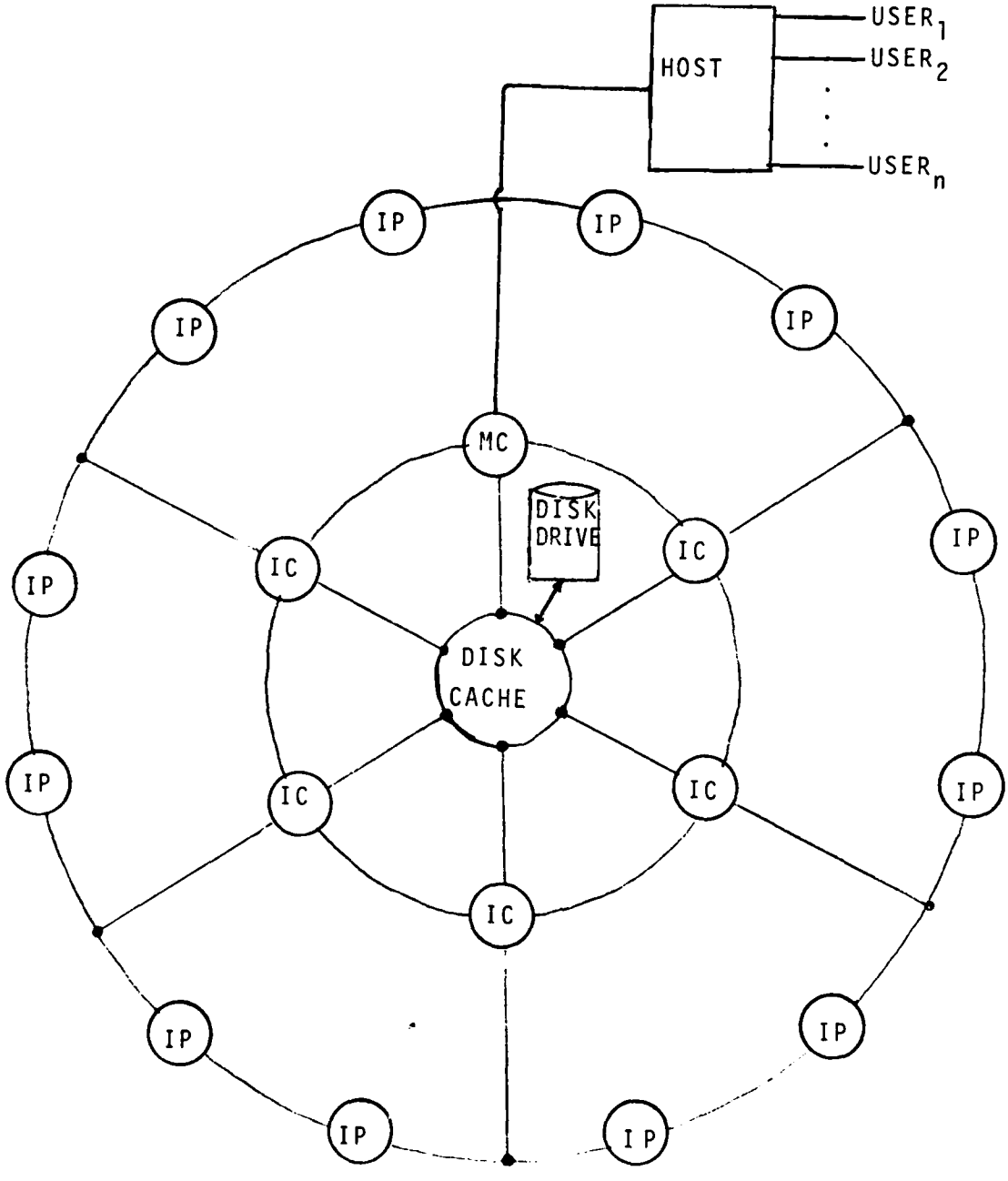


FIGURE 4.1

A DATA-FLOW DATABASE MACHINE CONFIGURATION

cache is to divide it among the ICs according to the number of IPs each is controlling. When an IC fills its segment of the disk cache, pages will be swapped out to disk. Thus, the IC local memory, the disk cache, and the mass storage devices form a three-level storage hierarchy.

IPs are responsible for executing instruction packets which are placed on the outer ring by the ICs. When an IP receives an instruction packet addressed to it, it performs the operation specified in the packet and then produces an output packet. The IP then places the output packet on the outer ring and sends it to the IC which is responsible for controlling the subsequent operation in the query tree. Thus, the IPs and the outer ring form a distributed distribution network for result packets.

The inner ring, as has been discussed above, is used exclusively for distribution of instructions and other control messages by the MC. Since the messages required for such activities are small and limited in number, a bandwidth of 1-2 million bits per second (Mbps) should be sufficient.

The outer ring, on the other hand, is used for distribution of instructions and result packets by the ICs and IPs. Figure 4.2 represents the bandwidth requirements of DIRECT [4] with page-level granularity for the test data described in Section 3.2. The bandwidth for each of the different processor levels was obtained by dividing the total number of bytes transferred by the execution time of the benchmark containing ten queries. Thus, the bandwidth values represent average values and not peak load values. (The bandwidth requirement for 70 processors

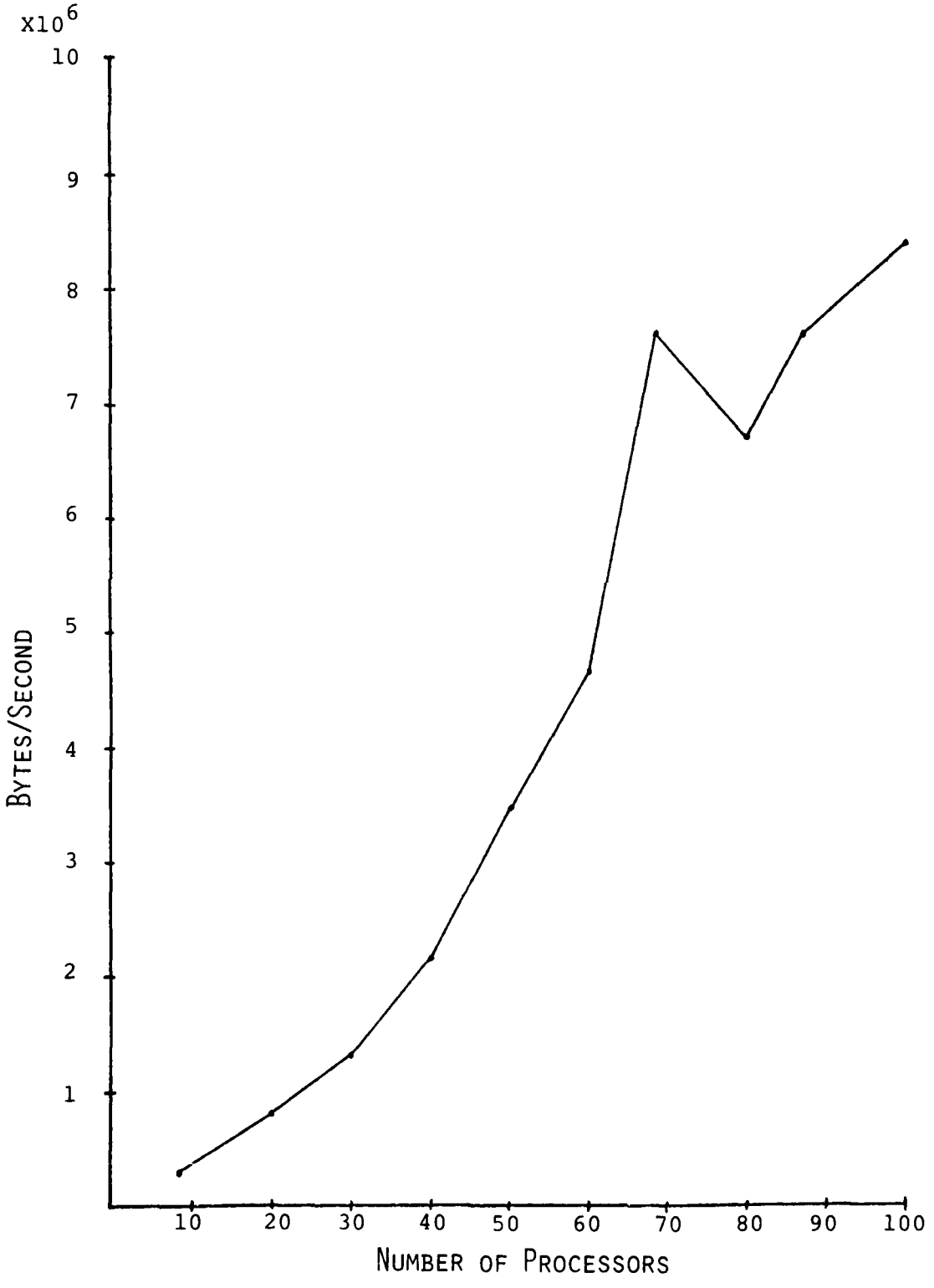


FIGURE 4.2

BANDWIDTH REQUIREMENTS OF DIRECT

represents a scheduling anomaly which is also reflected in Figure 3.1).

There are several possible technologies for the ring organization which we intend to investigate. One possible technology is that which is used in the Distributed Loop Computer Network [14]. This network employs a technique known as shift-register insertion and can handle the transmission of variable length messages. A ring bandwidth of up to 40Mbps can be obtained in this fashion. Some alternatives are loops constructed using either fiber optic technology or broadband coaxial technology. Fiber optics can support bandwidths of 400 Mbps [15] and should be commercially available in the next 5-10 years. Broadband coaxial technology is claimed to be capable of 100 Mbps.

4.2. Instruction Control and Execution

When an instruction is assigned to an IC it can be in one of two states. If the instruction's operands are source relations in the database, then the instruction is ready to be executed. In this case the MC will also send to the IC a page table describing each operand. Otherwise, if the instruction is not enabled, the IC will first create a page table for each operand of the instruction and then wait for pages of the source operand(s) to arrive from IPs being controlled by another IC. As pages (which may not be full) arrive, they are compressed to form full pages [3] and then stored in the IC's local memory or its segment of the disk cache.

When an IC is ready to initiate the execution of an instruc-

tion (i.e. at least one page of each operand is present), the IC first sends a control packet to the MC which requests an initial allocation of IPs and disk cache page frames. If the requested allocation cannot be fully satisfied, the MC will respond with a list of the IPs and page frames which are currently available. When another instruction has terminated, the MC will send the remaining requested resources to the IC.

The packet format of instruction packets sent by an IC to one of its IPs is shown in Figure 4.3. The destination of the packet is controlled by the IPid field. It is important to note that since packets are not fixed length, it is possible for the IC to send varying size data pages as source operands. In this way maximal concurrency can be achieved while the bandwidth requirements of the communications medium are minimized.

Upon receipt the IP applies the operation code to the data pages contained in the packet. Tuples of the result relation are placed by the IP in an internal buffer. The IP informs the controlling IC that it is done by sending it a control packet (Figure 4.4). The IC can respond by either sending additional packets or by releasing the IP. When the IP's internal buffer fills or when the flush-when-done flag of the instruction packet is on, the IP sends the contents of its buffer in a result packet (Figure 4.5) to the destination IC specified in the instruction packet. The controlling IC will turn the flush-when-done flag on when it expects the outgoing packet to be the last one which will be sent to the IP.

When an IP first receives an instruction packet for a join

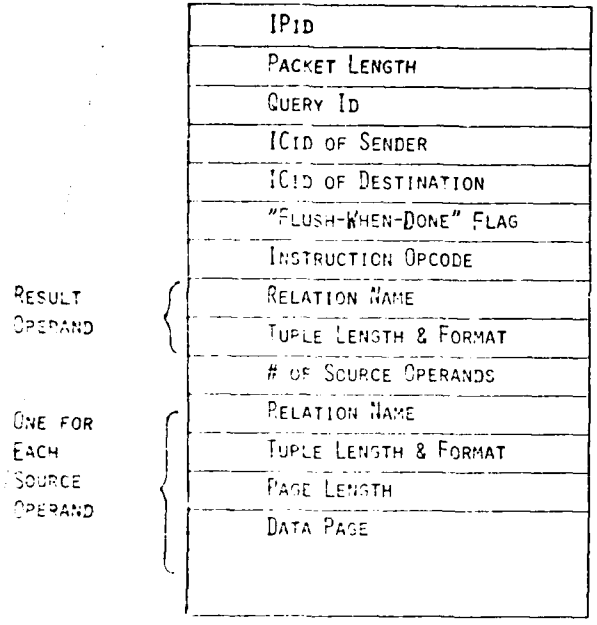


FIGURE 4.3

INSTRUCTION PACKET FORMAT

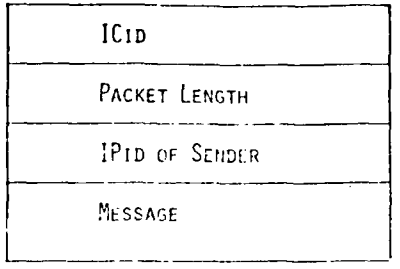


FIGURE 4.4

CONTROL PACKET FORMAT

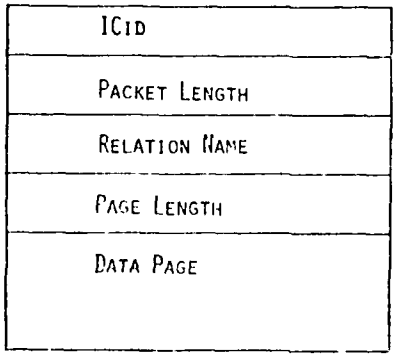


FIGURE 4.5

RESULT PACKET FORMAT

operation, it sets up an "inner-relation control" (IRC) vector with one entry for each page of the inner relation. (Initially this vector will have only one entry, but the vector will grow as execution of the instruction progresses.) After the IP has joined the first page of the outer relation with the first page of the inner relation (the two operands in the packet), the IP will send a "done" control packet to the controlling IC. Included in this packet will be a request for the second page of the inner relation. The IC responds to this request by broadcasting the requested page to all IPs which are executing the join. (An IP can determine if a broadcast packet is meant for it by examining the Query ID field of the packet). Subsequent requests for the same page which are received by the IC "soon" afterwards can be ignored.

Each IP which receives the broadcast packet can be in one of several states. If the IP has already sent or is about to send a request for the same page to the IC, then the IP can proceed to join the new page of the inner relation with its current page of the outer and update its IRC vector appropriately. If the IP does not have room in its local memory for the broadcast page, it will ignore the packet. However, the following scenario may occur. Because its local buffer is full, an IP ignores page i of the inner relation. When broadcast page i+1 is received (before it or page i has been solicited by the IP), the IP will read page i+1 and use it as an operand page. This situation can continue until a packet is received which indicates that this is the last page of the inner relation. At this point each IP will examine

its IRC vector and then proceed to request those pages which it missed. When the IP has joined the current page of the outer relation with all the pages of the inner relation, it will first zero its IRC vector and then signal the IC that it is ready for another page of the outer relation which has not yet been distributed to an IP. In this way message traffic on the outer ring is minimized and yet correct operation of the join can be guaranteed.

5. Conclusions and Future Research

In this paper we have presented the use of data-flow machine techniques for the processing of relational algebra queries. The performance of two multiprocessor join algorithms was analyzed and it was shown that the nested-loops algorithm is generally superior to the sort-merge algorithm. We have also discussed alternative operand granularities for data-flow database machines and have demonstrated that page-level granularity is the best choice for optimum system performance. A preliminary design for a data-flow database machine which utilizes page-level granularity and supports distributed control of instruction execution has been described.

There are several features of our proposed design with which we are not completely satisfied and which warrant further investigation. In particular, we feel that it should be possible to route some of the data pages which are produced by IPs directly from one IP to another without first sending the page to an IC. Thus, instruction execution control would be distributed further.

If such an approach could be successfully designed and implemented message traffic on the outer ring could be further reduced. There appears, however, to be a tradeoff between decreased message traffic and increased IP complexity which needs further examination before the correct approach can be chosen.

Another area of research which we intend to pursue is concurrency control mechanisms for data-flow database machines. In our current design, the MC is responsible for all concurrency control. We intend to investigate a distributed mechanism in which the ICs and not the MC would be responsible for concurrency control. One can view a data-flow database machine (such as described in Section 4.0) as a "local" distributed database system in which the ICs correspond to distributed centers of query execution and control. As a first step we intend to examine the mechanisms which have been proposed for concurrency control in distributed database systems to see if they are applicable. We intend to evaluate the performance of each of these algorithms to determine how each performs in a "local" environment. Then based on our findings we will either adopt one of the existing algorithms or attempt to develop a new algorithm which takes advantage of the "local" nature of the ICs.

While the ring architecture we have proposed seems to satisfy the organizational requirements for data-flow database machines, the requirement for a high bandwidth communications medium may not be realistic for a large number of IPs. The other MIMD database machines which have been proposed have depended on processor-memory interconnections in complexity of $O(n^2)$ [1,2] to

O(nlogn) [16-17]. We feel that the O(n) nature of the ring organization is the best if high (100 Mbit) bandwidth rings become available. However, we intend to investigate other processor interconnection strategies for data-flow database machines which satisfy the requirements specified in Section 4.0 yet which can be constructed from existing technologies.

6. Acknowledgements

Kevin Wilkinson's diligence and patience in listening and commenting on ideas is much appreciated. We also wish to thank Dina Friedland for her help in the analysis of the parallel join algorithms.

7. REFERENCES

1. DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," Proceedings of the 5th Annual Symposium on Computer Architecture, April 1978, pp. 182-189.

2. DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers, June 1979, pp. 395-406. Also: Computer Sciences Technical Report #325, Univ. of Wisconsin, June 1978.

3. DeWitt, D.J., "Query Execution in DIRECT", Proceedings of the ACM-SIGMOD 1979 International Conference of Management of Data, May 1979, pp 13-22.

4. Boral, H., and D.J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," submitted to ACM Transactions on Database Systems. Also Computer Sciences Technical Report No. 368, University of Wisconsin, October 1979.

5. Blasgen, M.W., and K.P. Eswaran, "Storage and Access in Relational Data Bases," IBM System Journal Vol. 16, No. 4, 1977, pp. 363-378.

6. Preparata, F.P., "New parallel-sorting schemes," IEEE

Transactions on Computers, Vol. C-27, July 1978, pp. 669-673.

7. Dennis, J.B., and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor,"The 2nd Annual Symposium on Computer Architecture: Conference Proceedings, January 1975, pp.126-132.

8. Arvind, and K.P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Information Processing 77: Proceedings of IFIP Congress 77, (B. Gilchrist, Ed.), August 1977, pp 849-853.

9. Davis, A.L., "The Architecture of DDM1: A Recursively Structured Data Driven Machine," Proceedings of the 5th Annual Symposium on Computer Architecture, April 1978, pp. 210-215.

10. Rumbaugh, J.E., "A Data Flow Multiprocessor," IEEE Transactions on Computers, February 1977, pp. 138-146.

11. Dennis, J.B., and K.S. Weng, "An Abstract Implementation for Concurrent Computation with Streams," Proceedings of the 1979 International Conference on Parallel Processing, August 1979, pp. 35-45.

12. Smith, J.M., and P. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," CACM Vol. 18, No. 10, October 1975, pp. 568-579.

13. Yao, S. Bing, "Optimization of Query Evaluation Algorithms," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979, pp. 133-155.

14. Liu, M.T., and C.C. Reames, "A Loop Network for the Simultaneous Transmission of Variable Length Messages," Proceedings of the 2nd Annual Conference on Computer Architecture, January 1975, pp. 7-12.

15. Frazer, W.D., "Potential Technology Implications for Computers and Telecommunications in the 1980s," IBM Systems Journal, Vol. 18, No. 2, 1979, pp. 333-347.

16. Hsiao, D.K., and J. Memon, "The Post Processing Functions of a Database Computer," Computer and Information Science Technical Report OSU-CISRC-TR-79-6, Ohio State University, July 1979.

17. Despain, A.M., and D.A. Patterson, "X-Tree: A Tree Structured Multi-Processor Computer Architecture," Proceedings of the 5th Annual Symposium on Computer Architecture, April 1978, pp. 144-151.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 2058	2. GOVT ACCESSION NO. AD-A086 379	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DESIGN CONSIDERATIONS FOR DATA-FLOW DATABASE MACHINES		5. TYPE OF REPORT & PERIOD COVERED Summary Report - no specific reporting period 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Haran Boral and David J. DeWitt		8. CONTRACT OR GRANT NUMBER(s) DAAG29-79-C-0165 DAAG29-75-C-0024 MCS78-01721
9. PERFORMING ORGANIZATION NAME AND ADDRESS Mathematics Research Center, University of Wisconsin 610 Walnut Street Madison, Wisconsin 53706		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 3 - Numerical Analysis and Computer Science
11. CONTROLLING OFFICE NAME AND ADDRESS See Item 18 below		12. REPORT DATE March 1980 13. NUMBER OF PAGES 31
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

National Science Foundation
Washington, D. C. 20550

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database management systems, relational databases, database machines, associative memories, parallel processing, query execution and optimization, dataflow machines.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents a discussion of the application of data-flow machine concepts to the design and implementation of database machines which execute relational algebra queries. We analyze the performance of multiprocessor nested-loops and sort-merge join algorithms and show that the nested-loops algorithm is generally superior. Three levels of operand granularity for data-flow database machines are introduced and compared using the nested-loops join algorithm. We demonstrate, that relation-level granularity is too coarse and that tuple-level granularity is too fine. The third level of granularity, a page of relation, is

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (continued)

shown to be the best choice from both hardware and software viewpoints. Finally a preliminary design for a data-flow database machine which utilizes page-level granularity and supports distributed control of instruction execution is presented.